Faire son propre OS (cpp) - le PDF

Credit : Je me sers en grande partie de cette chaine pour réaliser mon OS. Le pdf contient les informations utiles contenues dans ses vidéos.

Write your own Operating System

Share your videos with friends, family, and the world

https://www.youtube.com/@writeyourownoperatingsystem

sites utiles pour la création d’un OS :

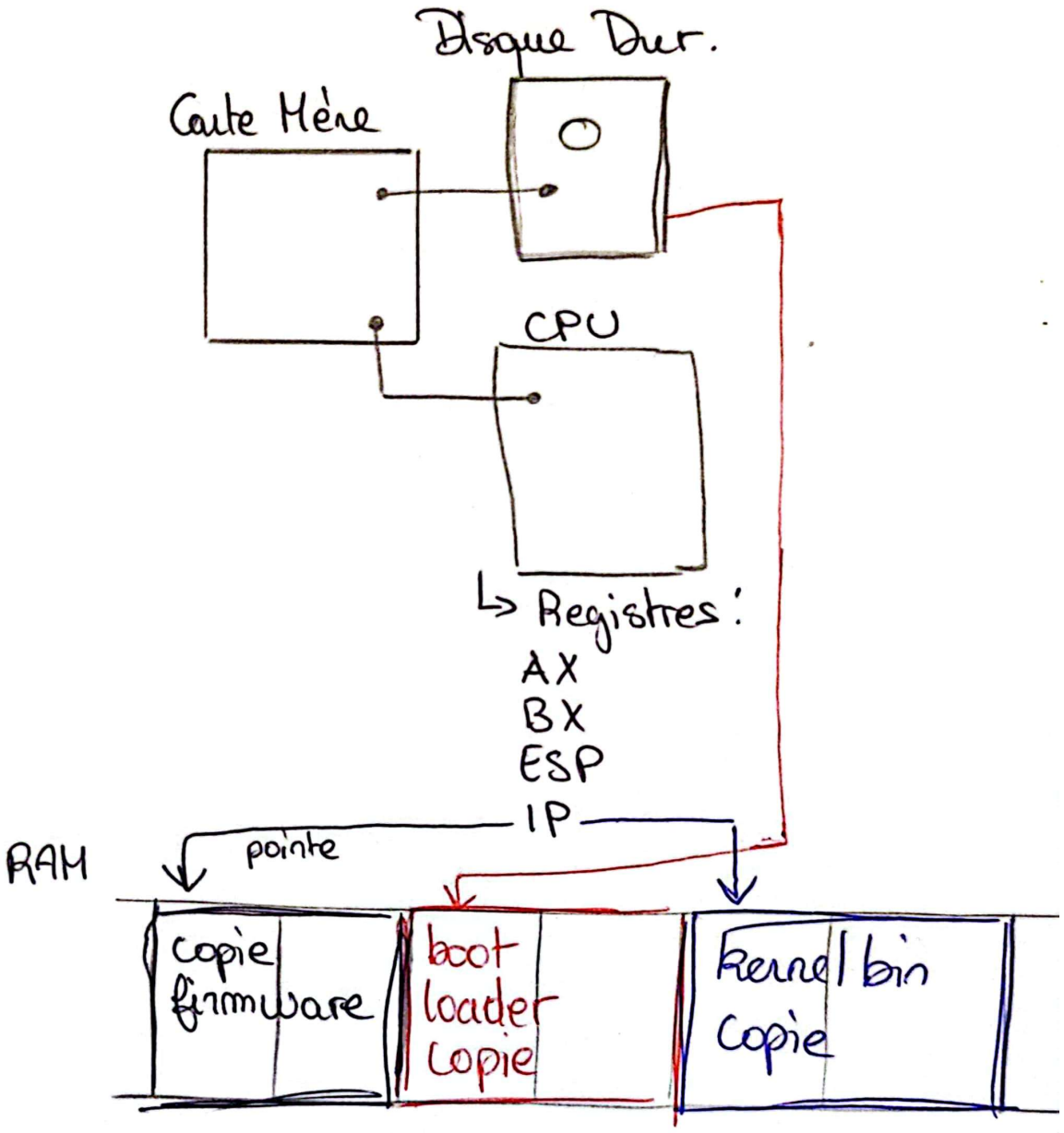
OSDev Wiki

Lowlevel.eu

Dépendances :

```
sudo apt-get install g++
                     binutils
                     libc6-dev-i386
```

Comprendre le démarrage d’un ordinateur



- Au démarrage, la carte mère va copier les données du BIOS (assembleur) dans la RAM, que l'on appelle le firmware (microcode, microprogramme). Il indique au CPU de pointer le registre IP (Instruction Pointer), ce qui permet au CPU de lire et exécuter les informations contenues dans la RAM. (flèche noire)

- Le firmware indique au CPU de lire le contenu du disque dur, qui sera chargé dans la RAM (boot loader). Ensuite, l'adresse du registre IP passe à celle du boot loader en RAM pour exécuter ce dernier. (flèche rouge)
- Le bootloader, complexe, gère les systèmes de fichiers ainsi que les partitions. Cela permet d'accéder au chemin `/boot/grub/grub.cfg`, qui contient le menu des entrées menant aux OS (en gros il permet de situer dans le DD les OS) (c'est la partie qui affiche les OS accessible lors du démarrage)
- Une fois choisie, le bootloader effectue une copie du noyau (ici kernelbin) dans la RAM et fait pointer le registre IP vers cette copie, ce qui permet à l'ordinateur de charger le système d'exploitation.

Problème 1 :

Le boot loader n'initialise pas le registre ESP (Stack Pointer). Cependant CPP nécessite d'avoir ce registre initialisé pour fonctionner correctement.

→ créer deux fichiers, `loader.s` en assembleur pour definir ESP, et sauter vers le deuxieme fichier CPP `kernel.cpp` qui contient l'OS

`loader.s` est compilé par GNU AS et donne `loader.o`, `kernel.cpp` qui contient l'OS.

est compilé par g++ et donne `kernel.o`

Problème 2 :

Les deux objets codés dans les langages différents doivent être combiné.

→ utilisation du programme LD. qui est un "linker" et donne un fichier kernel bin

nb : Le CPU par défaut au démarrage est compatible 32 bit, donc nous allons faire un noyau en 32 bit.

Problème 3 :

Le stack pointeur (négatif) peut pointer vers des emplacement mémoire de la RAM déjà utilisé et overwrite des informations, ce que l'on ne veut pas

→ il suffit d'allouer de la place afin de ne pas écrire (2mb suffisent)

Problème 4 :

Le boot loader ne va pas reconnaitre ce que l'on a fait comme un kernel, car il ne contient pas de metric number,

→ il suffit de le rajouter

IL EST : `0x1badb002`

La size de la RAM, le bootloader crée une structure et store a pointer vers cette structure dans le registre AX, et le metric number dans le registre BX

Problème 5 :

Comme nous codons en CPP, en dehors d'un OS.

```
#include <stdio.h>
```

N'existe pas. Ces bibliothèques ne sont pas contenues dans notre OS.

donc pas de dynamic linking, pas de memory management, pas de malloc etc…

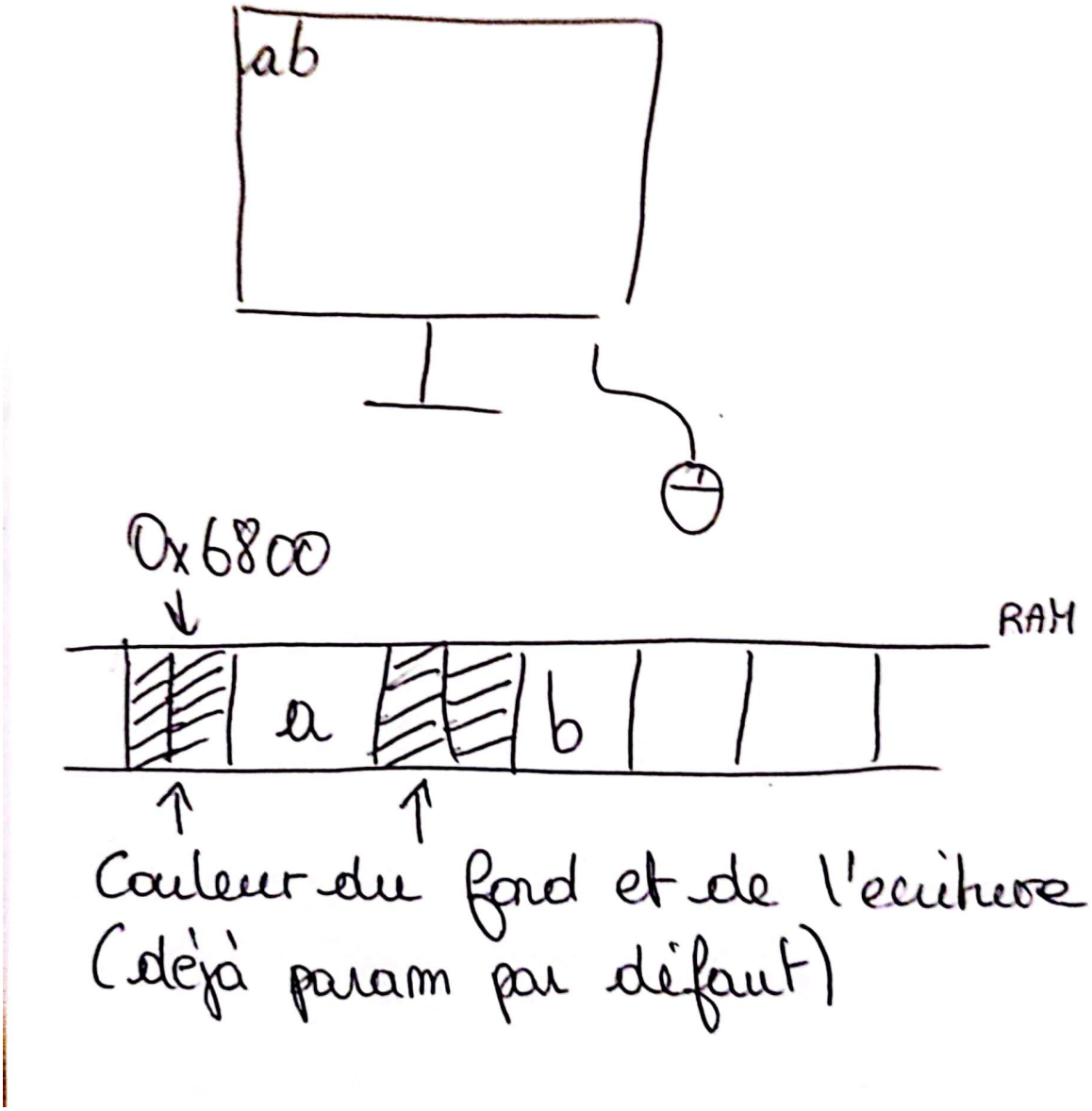
Solution : do it yourself :)

Nous devons donc écrire notre propre printf par exemple.

A savoir :

Lors qu'on écrit dans la ram à l'emplacement : 0xb8000

Ce que l'on écrit sur cette ligne sera affiché par la carte graphique



Ajouter l’os dans grub multiboot

et on ajoute ceci ! A ce moment où j’écris ce texte. il faut avoir grub, je ne sais pas si je vais coder un multiboot, car ma VM kali le contient déjà et est simple à mettre en place.

```
sudo vim /boot/grub/grub.cfg
```

```
### BEGIN MYKERNEL ###

menuentry 'Le nom de votre OS qui s affichera lors du démarrage.'{
  multiboot /boot/mykernel.bin
  boot
}

### END MYKERNEL ###

:w!
:q
```